# CS 61C:
# Great Ideas in Computer Architecture
# *RISC-V Instruction Formats*

# So What Is This "Frame Pointer" Business



- As a reminder, we shove all the C local variables etc on the stack...
  - Combined with space for all the saved registers
  - This is called the "activation record" or "call frame" or "call record"
- But a naive compiler may cause the stack pointer to bounce up and down during a function call
  - Can be a lot simpler to have a compiler do a bunch of pushes and pops when it needs a bit of temporary space: more so on a CISC rather than a RISC however
- Plus not all programming languages can store all activation records on the stack:
  - The use of lambda in Scheme, Python, Go, etc requires that some call frames are allocated on the heap since variables may last beyond the function call!

# So The Convention: Use S0 as a Frame Pointer (**fp**)



- At the start, save s0 and then have the Frame pointer point to one below the sp when you were called...
  - ```
    addi sp sp 20   # Initially grabbing 5 words of space
    sw ra sp 16     #
    sw fp sp 12     # save fp/s0
    addi fp sp 16   # Points to the start of this call record
    ...
    ```
- Now we can address local variables off the frame pointer rather than the stack pointer
  - Simplifies the compiler
    - Since it can now move the stack up and down easily
  - Simplifies the ***debugger***
    ```

# But Note...



- It isn't necessary in C...
  - Most C compilers has a -f-omit-frame-pointer option on most architectures
    - It just fubars debugging a bit
- So for our hand-written assembly, we will generally ignore the frame pointer
- The calling convention says it doesn't matter if you use a frame pointer or not!
  - It is just a callee saved register, so if you use it as a frame pointer...
    It will be preserved just like any other saved register
    But if you just use it as **`s0`**, that makes no difference!

# Stack Also For Local Variables...



- e.g.
  char[20] foo;
- Requires enough space on the stack
  - May need padding
- So then to pass foo to something in a0...
  - **`addi a0 sp`** ***`offset-for-foo-off-sp`***
  - **`addi a0 fp`** ***`offset-for-foo-off-fp`***
    - If you are using the frame pointer...

# A Richer C->RISC-V Translation Example



```
typedef struct list
    {void *car;
     struct list *cdr;} List;

List *map(List *src, void * (*f) (void *)){
  List *ret;
  if(!src) return 0;
  ret = (List *) malloc(sizeof(List);
  ret->car = (*f)(src->car);
  ret->cdr = map(src->cdr, f);
  return ret;
}
```

# What Do We Need To Keep Track Of?



- Both arguments are needed later:
  src and f
- Local variable ret
- We could use either saved registers or the stack...
  - But let us use saved registers:
    s0 for src
    s1 for f
    s2 for ret
  - Ends up being easier to keep track of
    - Not bother with the frame pointer...

# Preamble



```
map:
  addi sp sp -16    # Need to store 4 entries
  sw   ra 12(sp)
  sw   s0 8(sp)
  sw   s1 4(sp)
  sw   s2 0(sp)
  mv   s0 a0        # save src
  mv   s1 a1        # save f
```

# Body...



```
  bne  a0 x0 map_skip_if
  mv   a0 x0               # return 0
  j    map_return
map_skip_if:
  li   a0 8                # sizeof(List)
  jal  ra malloc           # call malloc
  mv   s2 a0               # save ret
  lw   a0 0(s0)            # src->car
  jalr ra s1               # call f
  sw   a0 0(s2)            # ret->car assigned
```

# Body and postamble

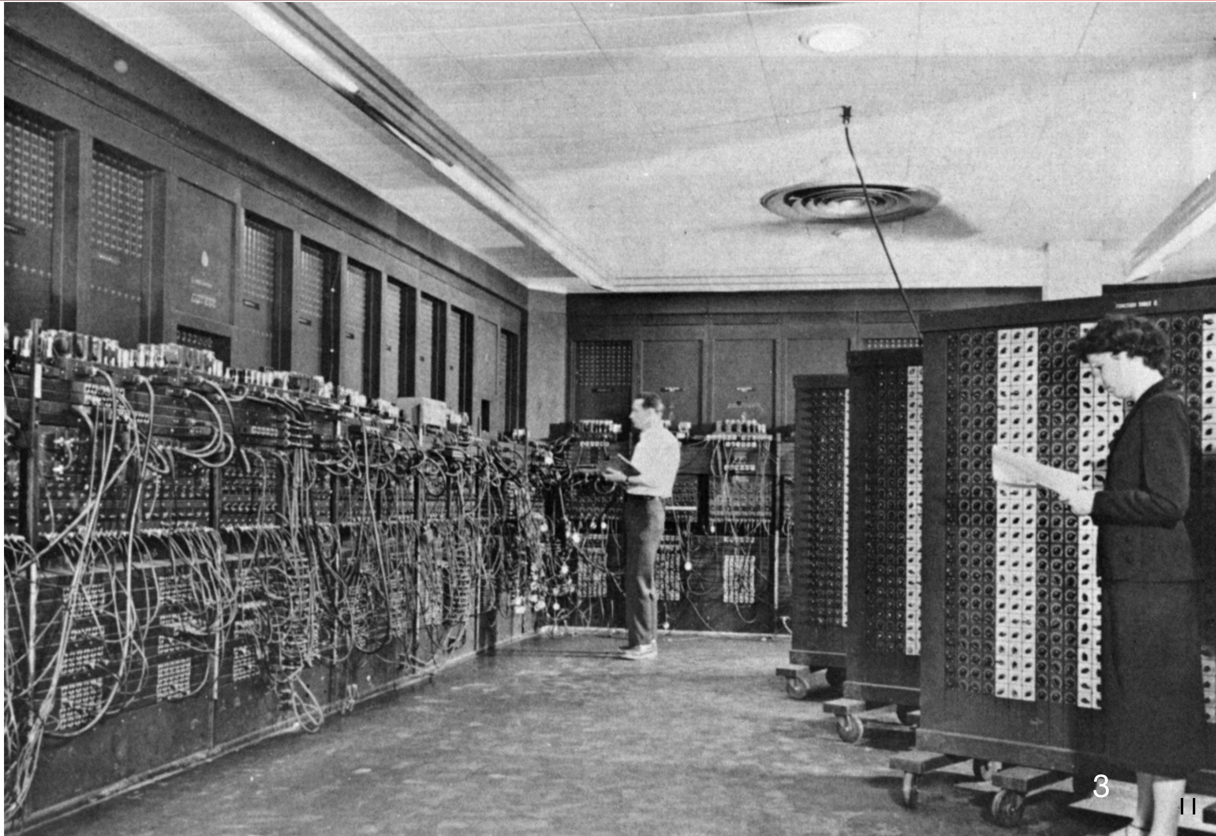


```
  lw a0 4(s0)       # src->cdr
  mv a1 s1          # f
  jal ra map        # recursive call
  sw a0 4(s2)       # store to ret->cdr
  mv a0 s2
map_return:
  lw ra 12(sp)
  lw s0 8(sp)
  lw s1 4(sp)
  lw s2 0(sp)
  addi sp sp 16
  jalr x0 ra
```

# ENIAC (U.Penn., 1946)
# First Electronic General-Purpose Computer



- Blazingly fast (multiply in 2.8ms!)
  - 10 decimal digits x 10 decimal digits
- But needed 2-3 days to setup new program, as programmed with patch cords and switches



# Big Idea: Stored-Program Computer



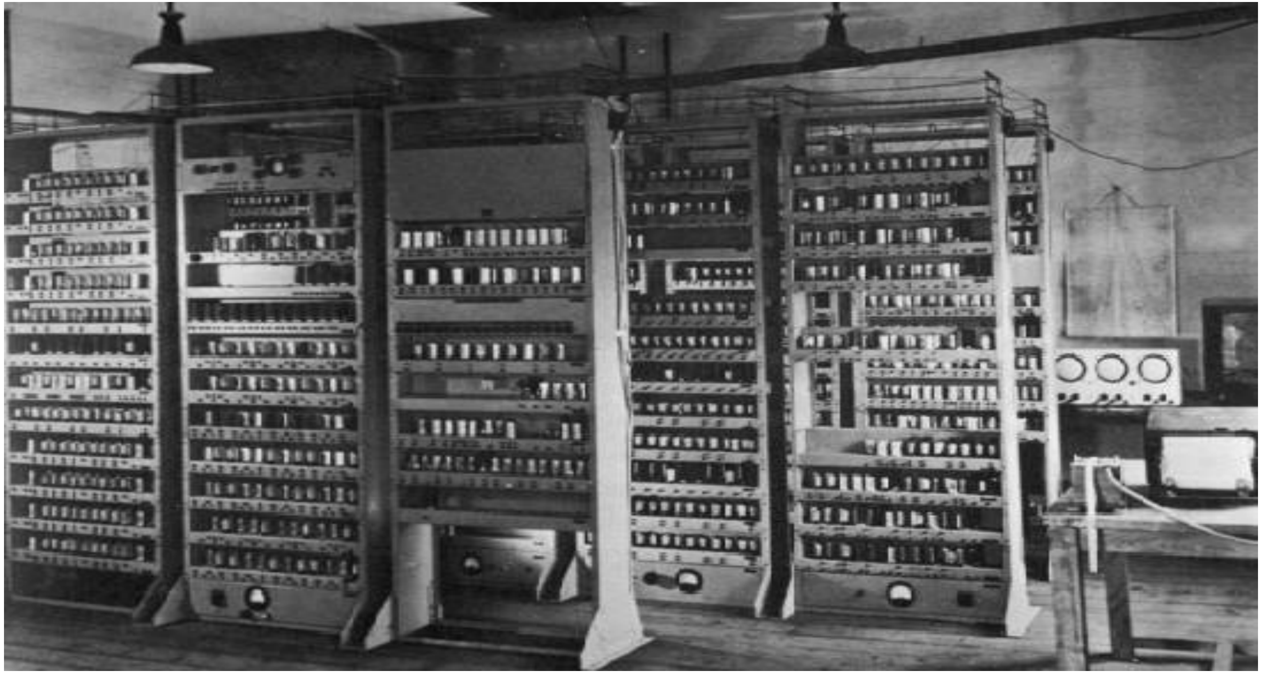
- Instructions are represented as bit patterns - can think of these as numbers
- Therefore, entire programs can be stored in memory to be read or written just like data
- Can reprogram quickly (seconds), don't have to rewire computer (days)
- Known as the "von Neumann" computers after widely distributed tech report on EDVAC project
  - Wrote-up discussions of Eckert and Mauchly
  - Anticipated earlier by Turing and Zuse

First Draft of a Report on the EDVAC
by
John von Neumann
Contract No. W–670–ORD–4926
Between the
United States Army Ordnance Department
and the
University of Pennsylvania
Moore School of Electrical Engineering
University of Pennsylvania
June 30, 1945

# EDSAC (Cambridge, 1949)
# First General Stored-Program Computer



- Programs held as “numbers” in memory
- 35-bit binary 2’s complement words



# Consequence #1: Everything Has a Memory Address



- Since all instructions and data are stored in memory, everything has a memory address: instructions, data words
  - Both branches and jumps use these
- C pointers are just memory addresses: they can point to anything in memory
- One register keeps address of instruction being executed: "Program Counter" (`PC`)
  - Basically a pointer to memory
  - Intel calls it Instruction Pointer (a better name)

# Consequence #2: Binary Compatibility



- Programs are distributed in binary form
  - Programs bound to specific instruction set
  - Different version for phones and PCs, etc.
- New machines in the same family want to run old programs (“binaries”) as well as programs compiled to new instructions
- Leads to “backward-compatible” instruction set evolving over time
  - Selection of Intel 8088 in 1981 for 1st IBM PC is major reason latest PCs still use 80x86 instruction set; could still run program from 1981 PC today

# Instructions as Numbers (1/2)



- Most data we work with is in words (32-bit chunks):
  - Each register is a word
  - **`lw`** and **`sw`** both access memory one word at a time
- So how do we represent instructions?
  - Remember: Computer only represents 1s and 0s, so assembler string "**`add x10,x11,x0`**" is meaningless to hardware
  - RISC-V seeks simplicity: since data is in words, make instructions be fixed-size 32-bit words also
    - Same 32-bit instruction definitions used for RV32, RV64, RV128

# Instructions as Numbers (2/2)



- Divide 32-bit instruction word into "fields"
- Each field tells processor something about instruction
- We could define different set of fields for each instruction, but RISC-V seeks simplicity, so group possible instructions into six basic types of instruction formats:
  - **R-format** for register-register arithmetic/logical operations
  - **I-format** for register-immediate ALU operations and loads
  - **S-format** for stores
  - **B-format** for branches
  - **U-format** for 20-bit upper immediate instructions
  - **J-format** for jumps

# Summary of RISC-V Instruction Formats



| 31 | 30 25 | 24 21 | 20 | 19 15 | 14 12 | 11 8 | 7 | 6 0 | |
|---|---|---|---|---|---|---|---|---|---|
| funct7 | | rs2 | | rs1 | funct3 | rd | | opcode | R-type |
| imm[11:0] | | | | rs1 | funct3 | rd | | opcode | I-type |
| imm[11:5] | | rs2 | | rs1 | funct3 | imm[4:0] | | opcode | S-type |
| imm[12] | imm[10:5] | rs2 | | rs1 | funct3 | imm[4:1] | imm[11] | opcode | B-type |
| imm[31:12] | | | | | | rd | | opcode | U-type |
| imm[20] | imm[10:1] | | imm[11] | imm[19:12] | | rd | | opcode | J-type |

# R-Format Instruction Layout



Field's bit positions

| 31 - 25 | 24 - 20 | 19 - 15 | 14 - 12 | 11 - 7 | 6 - 0 |
|---|---|---|---|---|---|
| funct7 | rs2 | rs1 | funct3 | rd | opcode |
| 7 | 5 | 5 | 3 | 5 | 7 |

Name of field

Number of bits in field

- This example: 32-bit instruction word divided into six fields of varying numbers of bits each: 7+5+5+3+5+7 = 32
- In this case:
  - opcode is a 7-bit field that lives in bits 6-0 of the instruction
  - rs2 is a 5-bit field that lives in bits 24-20 of the instruction
  - etc.

# R-Format Instructions opcode/funct fields

| 31 — 25 | 24 — 20 | 19 — 15 | 14 — 12 | 11 — 7 | 6 — 0 |
|---|---|---|---|---|---|
| funct7 | rs2 | rs1 | funct3 | rd | opcode |
| 7 | 5 | 5 | 3 | 5 | 7 |

- `opcode`: partially specifies which instruction it is
  - Note: This field is equal to `0110011`$_{two}$ for all R-Format register-register arithmetic/logical instructions
- `funct7+funct3`: combined with **`opcode`**, these two fields describe what operation to perform

- Question: Why aren't **`opcode`** and **`funct7`** and **`funct3`** a single 17-bit field?
  – We'll answer this later

# R-Format Instructions register specifiers

| 31 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 |
|---|---|---|---|---|---|
| funct7 | rs2 | rs1 | funct3 | rd | opcode |
| 7 | 5 | 5 | 3 | 5 | 7 |

- Each register field (rs1, rs2, rd) holds a 5-bit unsigned integer (0-31) corresponding to a register number (**`x0-x31`**)
- **`rs1`** (Source Register #1): specifies register containing first operand
- **`rs2`** : specifies second register operand
- **`rd`** (Destination Register): specifies register which will receive result of computation

# R-Format Example



- RISC-V Assembly Instruction:

```
add  x18,x19,x10
```

| 31 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 |
|---|---|---|---|---|---|
| funct7 | rs2 | rs1 | funct3 | rd | opcode |
| 7 | 5 | 5 | 3 | 5 | 7 |
| 0000000 | 01010 | 10011 | 000 | 10010 | 0110011 |
| ADD | rs2=10 | rs1=19 | ADD | rd=18 | Reg-Reg OP |

# All RV32 R-format instructions

| funct7 | | | funct3 | | opcode | |
|---|---|---|---|---|---|---|
| 0000000 | rs2 | rs1 | 000 | rd | 0110011 | ADD |
| 0100000 | rs2 | rs1 | 000 | rd | 0110011 | SUB |
| 0000000 | rs2 | rs1 | 001 | rd | 0110011 | SLL |
| 0000000 | rs2 | rs1 | 010 | rd | 0110011 | SLT |
| 0000000 | rs2 | rs1 | 011 | rd | 0110011 | SLTU |
| 0000000 | rs2 | rs1 | 100 | rd | 0110011 | XOR |
| 0000000 | rs2 | rs1 | 101 | rd | 0110011 | SRL |
| 0100000 | rs2 | rs1 | 101 | rd | 0110011 | SRA |
| 0000000 | rs2 | rs1 | 110 | rd | 0110011 | OR |
| 0000000 | rs2 | rs1 | 111 | rd | 0110011 | AND |

Encoding in funct7 + funct3 selects particular operation

# I-Format Instructions



- What about instructions with immediates?
  - Ideally, RISC-V would have only one instruction format (for simplicity): unfortunately, we need to compromise
  - 5-bit field only represents numbers up to the value 31: would like immediates to be much larger
- Define another instruction format that is mostly consistent with R-format
  - Note: if instruction has immediate, then uses at most 2 registers (one source, one destination)

# I-Format Instruction Layout

| 31 ... 20 | 19 ... 15 | 14 ... 12 | 11 ... 7 | 6 ... 0 |
|---|---|---|---|---|
| imm[11:0] | rs1 | funct3 | rd | opcode |
| 12 | 5 | 3 | 5 | 7 |

- Only one field is different from R-format, rs2 and funct7 replaced by 12-bit signed immediate, **`imm[11:0]`**
- Remaining field format (rs1, funct3, rd, opcode) same as before
- imm[11:0] can hold values in range [$-2048_{ten}$ , $+2047_{ten}$ ]
- Immediate is ***always*** sign-extended to 32-bits before use in an arithmetic operation
- We'll later see how to handle immediates > 12 bits

# I-Format Example



- RISC-V Assembly Instruction:

```
addi  x15,x1,-50
```

| 31 — 20 | 19 — 15 | 14 — 12 | 11 — 7 | 6 — 0 |
|---|---|---|---|---|
| imm[11:0] | rs1 | funct3 | rd | opcode |
| 12 | 5 | 3 | 5 | 7 |

| | | | | |
|---|---|---|---|---|
| 111111001110 | 00001 | 000 | 01111 | 0010011 |
| imm=-50 | rs1=1 | ADD | rd=15 | OP-Imm |

# All RV32 I-format Arithmetic/Logical Instructions

| imm | | | funct3 | | opcode | |
|---|---|---|---|---|---|---|
| imm[11:0] | | rs1 | 000 | rd | 0010011 | ADDI |
| imm[11:0] | | rs1 | 010 | rd | 0010011 | SLTI |
| imm[11:0] | | rs1 | 011 | rd | 0010011 | SLTIU |
| imm[11:0] | | rs1 | 100 | rd | 0010011 | XORI |
| imm[11:0] | | rs1 | 110 | rd | 0010011 | ORI |
| imm[11:0] | | rs1 | 111 | rd | 0010011 | ANDI |
| 0000000 | shamt | rs1 | 001 | rd | 0010011 | SLLI |
| 0000000 | shamt | rs1 | 101 | rd | 0010011 | SRLI |
| 0100000 | shamt | rs1 | 101 | rd | 0010011 | SRAI |

One of the higher-order immediate bits is used to distinguish “shift right logical” (SRLI) from “shift right arithmetic” (SRAI)

“Shift-by-immediate” instructions only use lower 5 bits of the immediate value for shift amount (can only shift by 0-31 bit positions)

# Administrivia



- Project 2: Due this Saturday
  - But you have slip days
  - Project PARTAAAYYYS:
    Wednesday 7-9, 273/275/277 Soda Hall
    Thursday 7-9, 540 Cory AB
- Midterm 1: 2/19, 8-10 pm
  - Room assignments will be posted on Piazza shortly
  - DSP and other accommodations will be handled in Piazza as well
- MT1 Reviews
  - HKN: Saturday Feb 16th, 4-7 pm, 306 Soda
  - Course Staff: Sunday Feb 17th, 9-12am, 306 Soda
  - CSM: Sunday Feb 17th, 3-4:30 (on C), 4:30-6 (on RISC-V), 310 Soda

# Clicker Question: How Was Project 1.1?



- A: 😃
  B: 🤨
  C: 😟
  D: 💩
  E: 🤷‍♂️

# Load Instructions are also I-Type



| 31 – 20 | 19 – 15 | 14 – 12 | 11 – 7 | 6 – 0 |
|---|---|---|---|---|
| imm[11:0] | rs1 | funct3 | rd | opcode |
| 12 | 5 | 3 | 5 | 7 |
| offset[11:0] | base | width | dest | LOAD |

- This is very similar to the add-immediate operation but used to create address not to create final result
- The value loaded from memory is stored in register rd

# I-Format Load Example



- RISC-V Assembly Instruction:

```
lw x14, 8(x2)
```

| 31 – 20 | 19 – 15 | 14 – 12 | 11 – 7 | 6 – 0 |
|---|---|---|---|---|
| imm[11:0] | rs1 | funct3 | rd | opcode |
| 12 | 5 | 3 | 5 | 7 |

| | | | | |
|---|---|---|---|---|
| 000000001000 | 00010 | 010 | 01110 | 0000011 |
| imm=+8 | rs1=2 | LW | rd=14 | LOAD |

# All RV32 Load Instructions



| | | | | | |
|---|---|---|---|---|---|
| imm[11:0] | rs1 | 000 | rd | 0000011 | LB |
| imm[11:0] | rs1 | 001 | rd | 0000011 | LH |
| imm[11:0] | rs1 | 010 | rd | 0000011 | LW |
| imm[11:0] | rs1 | 100 | rd | 0000011 | LBU |
| imm[11:0] | rs1 | 101 | rd | 0000011 | LHU |

funct3 field encodes size and signedness of load data

- LBU is “load unsigned byte”
- LH is “load halfword”, which loads 16 bits (2 bytes) and sign-extends to fill destination 32-bit register
- LHU is “load unsigned halfword”, which zero-extends 16 bits to fill destination 32-bit register
- There is no LWU in RV32, because there is no sign/zero extension needed when copying 32 bits from a memory location into a 32-bit register

# S-Format Used for Stores

| 31 — 25 | 24 — 20 | 19 — 15 | 14 — 12 | 11 — 7 | 6 — 0 |
|---|---|---|---|---|---|
| imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode |
| 7 | 5 | 5 | 3 | 5 | 7 |
| offset[11:5] | src | base | width | offset[4:0] | STORE |

- Store needs to read two registers, **`rs1`** for base memory address, and **`rs2`** for data to be stored, as well as need immediate offset!
- Can't have both **`rs2`** and immediate in same place as other instructions!
- Note that stores don't write a value to the register file, ***no rd***!
- RISC-V design decision is move low 5 bits of immediate to where rd field was in other instructions – keep **`rs1`**/**`rs2`** fields in same place.

# Keeping Registers always in the Same Place...



- The critical path for ***all operations*** includes fetching values from the registers
- By always placing the read sources in the same place, the register file can read without hesitation
  - If the data ends up being unnecessary (e.g. I-Type), it can be ignored
- Other RISCs have had slightly different encodings
  - Necessitating the logic to look at the instruction to determine which registers to read
- Example of one of the (many) little tweaks done in RISC-V to make things work better

# S-Format Example



- RISC-V Assembly Instruction:

```
sw x14, 8(x2)
```

| 31 — 25 | 24 — 20 | 19 — 15 | 14 — 12 | 11 — 7 | 6 — 0 |
|---|---|---|---|---|---|
| imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode |
| 7 | 5 | 5 | 3 | 5 | 7 |
| offset[11:5] | src | base | width | offset[4:0] | STORE |

| | | | | | |
|---|---|---|---|---|---|
| 0000000 | 01110 | 00010 | 010 | 01000 | 0100011 |
| offset[11:5] =0 | rs2=14 | rs1=2 | SW | offset[4:0] =8 | STORE |

| | |
|---|---|
| 0000000 | 01000 |

combined 12-bit offset = 8

# All RV32 Store Instructions



| | | | | | | |
|---|---|---|---|---|---|---|
| imm[11:5] | rs2 | rs1 | 000 | imm[4:0] | 0100011 | SB |
| imm[11:5] | rs2 | rs1 | 001 | imm[4:0] | 0100011 | SH |
| imm[11:5] | rs2 | rs1 | 010 | imm[4:0] | 0100011 | SW |

# RISC-V Conditional Branches



- E.g., **`BEQ x1, x2, Label`**
- Branches read two registers but don't write a register (similar to stores)
- How to encode label, i.e., where to branch to?

# Branching Instruction Usage



- Branches typically used for loops (`if-else`, `while`, `for`)
  - Loops are generally small (< 50 instructions)
  - Function calls and unconditional jumps handled with jump instructions (J-Format)
- **Recall:** Instructions stored in a localized area of memory (Code/Text)
  - Largest branch distance limited by size of code
  - Address of current instruction stored in the program counter (PC)

# PC-Relative Addressing



- PC-Relative Addressing: Use the `immediate` field as a two's-complement *offset* relative to PC
  - Branches generally change the PC by a small amount
  - Could specify ± $2^{11}$ addresses offset from the PC
- To improve the reach of a single branch instruction, *in principle*, could multiply the offset by four bytes before adding to PC (*instructions are 4 bytes and word aligned*).
- This would allow one branch instruction to reach ± $2^{11}$ × 32-bit instructions either side of PC
  - Four times greater reach than using byte offset
  - However …

# RISC-V Feature, n×16-bit instructions



- Extensions to RISC-V base ISA support 16-bit compressed instructions and also variable-length instructions that are multiples of 2-Bytes in length
- To enable this, RISC-V always scales the branch offset by 2 bytes - even when there are no 16-bit instructions
- (This means for us, the low bit of the stored immediate value will always be 0)
- Reduces branch reach by half:
- RISC-V conditional branches can only reach $\pm 2^{10} \times$ 32-bit instructions either side of PC

# Branch Calculation



- If we don't take the branch:

  `PC = PC + 4` (i.e., next instruction)

- If we do take the branch:

  `PC = PC + immediate*2`

- **Observations:**
  - `immediate` is number of instructions to jump (remember, specifies words) either forward (+) or backwards (–)

# RISC-V B-Format for Branches



| 31 | 30 … 25 | 24 … 20 | 19 … 15 | 14 … 12 | 11 … 8 | 7 | 6 … 0 |
|---|---|---|---|---|---|---|---|
| imm[12] | imm[10:5] | rs2 | rs1 | funct3 | imm[4:1] | imm[11] | opcode |
| 1 | 6 | 5 | 5 | 3 | 4 | 1 | 7 |

- B-format is mostly same as S-Format, with two register sources (rs1/rs2) and a 12-bit immediate
- But now immediate represents values -4096 to +4094 in 2-byte increments
- The 12 immediate bits encode *even* 13-bit signed byte offsets (lowest bit of offset is always zero, so no need to store it)

# Branch Example, determine offset



- RISC-V Code:

```
Loop: beq  x19,x10,End
      add  x18,x18,x10
      addi x19,x19,-1
      j    Loop
End:  # target instruction
```

1
2
3
4

Count instructions from branch

- Branch offset = 4×32-bit instructions = 16 bytes
- (Branch with offset of 0, branches to itself)

# Branch Example, encode offset



- RISC-V Code:

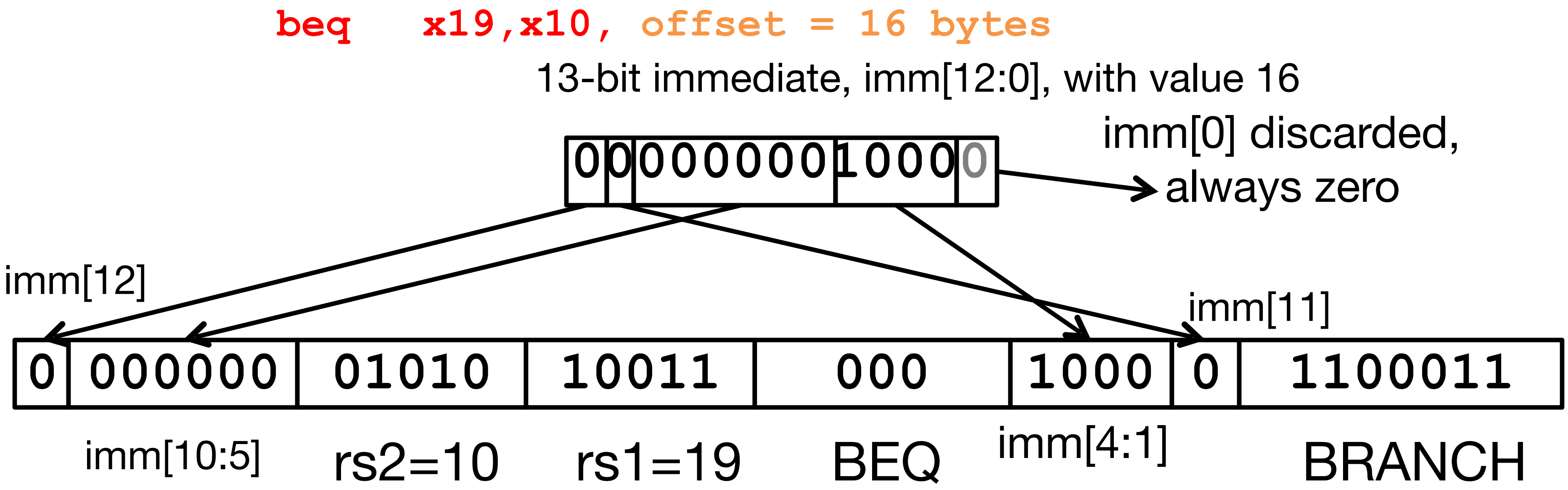
```
Loop: beq   x19,x10,End
      add   x18,x18,x10
      addi  x19,x19,-1
      j     Loop
End: # target instruction
```

**offset = 16 bytes = 8x2**

| ??????? | 01010 | 10011 | 000 | ????? | 1100011 |
|---|---|---|---|---|---|
| imm | rs2=10 | rs1=19 | BEQ | imm | BRANCH |

# Branch Example, complete encoding





# All RISC-V Branch Instructions



| | | | | | | |
|---|---|---|---|---|---|---|
| imm[12\|10:5] | rs2 | rs1 | 000 | imm[4:1\|11] | 1100011 | BEQ |
| imm[12\|10:5] | rs2 | rs1 | 001 | imm[4:1\|11] | 1100011 | BNE |
| imm[12\|10:5] | rs2 | rs1 | 100 | imm[4:1\|11] | 1100011 | BLT |
| imm[12\|10:5] | rs2 | rs1 | 101 | imm[4:1\|11] | 1100011 | BGE |
| imm[12\|10:5] | rs2 | rs1 | 110 | imm[4:1\|11] | 1100011 | BLTU |
| imm[12\|10:5] | rs2 | rs1 | 111 | imm[4:1\|11] | 1100011 | BGEU |

# Questions on PC-addressing



- Does the value in branch immediate field change if we move the code?
  - If moving individual lines of code, then yes
  - If moving all of code, then no (because PC-relative offsets)
- What do we do if destination is > $2^{10}$ instructions away from branch?
  - Other instructions save us
  - 
```
beq x10,x0,far                      bne x10,x0,next
# next instr          →                 j    far
                              next: # next instr
```

# U-Format for “Upper Immediate” instructions



| 31 … 12 | 11 … 7 | 6 … 0 |
|---|---|---|
| imm[31:12] | rd | opcode |
| 20 | 5 | 7 |
| U-immediate[31:12] | dest | LUI |
| U-immediate[31:12] | dest | AUIPC |

- Has 20-bit immediate in upper 20 bits of 32-bit instruction word
- One destination register, rd
- Used for two instructions
  - LUI – Load  Upper Immediate
  - AUIPC – Add Upper Immediate to PC

# LUI to create long immediates



- LUI writes the upper 20 bits of the destination with the immediate value, and clears the lower 12 bits.
- Together with an ADDI to set low 12 bits, can create any 32-bit value in a register using two instructions (LUI/ADDI).

```
LUI x10, 0x87654      # x10 = 0x87654000
ADDI x10, x10, 0x321# x10 = 0x87654321
```

# One Corner Case



How to set 0xDEADBEEF?

```
LUI x10, 0xDEADB       # x10 = 0xDEADB000
ADDI x10, x10, 0xEEF# x10 = 0xDEADAEEF
```

ADDI 12-bit immediate is always sign-extended, if top bit is set, will subtract -1 from upper 20 bits

# Solution



How to set 0xDEADBEEF?

```
LUI x10, 0xDEADC       # x10 = 0xDEADC000
ADDI x10, x10, 0xEEF# x10 = 0xDEADBEEF
```

Pre-increment the value placed in upper 20 bits, if sign bit will be set on immediate in lower 12 bits.

Assembler pseudo-op handles all of this:

```
li x10, 0xDEADBEEF # Creates two instructions
```

# J-Format for Jump Instructions



| 31 | 30 … 21 | 20 | 19 … 12 | 11 … 7 | 6 … 0 |
|---|---|---|---|---|---|
| imm[20] | imm[10:1] | imm[11] | imm[19:12] | rd | opcode |
| 1 | 10 | 1 | 8 | 5 | 7 |
| | offset[20:1] | | | dest | JAL |

- JAL saves PC+4 in register rd (the return address)
  - Assembler "`j`" jump is pseudo-instruction, uses JAL but sets rd=x0 to discard return address
- Set PC = PC + offset (PC-relative jump)
- Target somewhere within $\pm 2^{19}$ locations, 2 bytes apart
  - $\pm 2^{18}$ 32-bit instructions
- Immediate encoding optimized similarly to branch instruction to reduce hardware cost

# Uses of JAL



```
# j pseudo-instruction
j Label = jal x0, Label # Discard return address

# Call function within 2^18 instructions of PC
jal ra, FuncName
```

# JALR Instruction (I-Format)



| 31 — 20 | 19 — 15 | 14 — 12 | 11 — 7 | 6 — 0 |
|---|---|---|---|---|
| imm[11:0] | rs1 | funct3 | rd | opcode |
| 12 | 5 | 3 | 5 | 7 |
| offset[11:0] | base | 0 | dest | JALR |

- JALR rd, rs, immediate
  - Writes PC+4 to rd (return address)
  - Sets PC = rs + immediate
  - Uses same immediates as arithmetic and loads
    - ***no*** multiplication by 2 bytes

# Uses of JALR



```
# ret and jr psuedo-instructions
ret = jr ra = jalr x0, ra, 0

# Call function at any 32-bit absolute address
lui  x1, <hi20bits>
jalr ra, x1, <lo12bits>

# Jump PC-relative with 32-bit offset
auipc x1, <hi20bits>    # Adds upper immediate value to
                        # and places result in x1
jalr x0, x1, <lo12bits> # Same sign extension trick needed
                        # as LUI
```

# Summary of RISC-V Instruction Formats



| 31 | 30 25 | 24 21 | 20 | 19 15 | 14 12 | 11 8 | 7 | 6 0 | |
|---|---|---|---|---|---|---|---|---|---|
| funct7 | | rs2 | | rs1 | funct3 | rd | | opcode | R-type |
| imm[11:0] | | | | rs1 | funct3 | rd | | opcode | I-type |
| imm[11:5] | | rs2 | | rs1 | funct3 | imm[4:0] | | opcode | S-type |
| imm[12] | imm[10:5] | rs2 | | rs1 | funct3 | imm[4:1] | imm[11] | opcode | B-type |
| imm[31:12] | | | | | | rd | | opcode | U-type |
| imm[20] | imm[10:1] | | imm[11] | imm[19:12] | | rd | | opcode | J-type |

# Complete RV32I ISA



| | | | | | | |
|---|---|---|---|---|---|---|
| imm[31:12] | | | | rd | 0110111 | LUI |
| imm[31:12] | | | | rd | 0010111 | AUIPC |
| imm[20\|10:1\|11\|19:12] | | | | rd | 1101111 | JAL |
| imm[11:0] | | rs1 | 000 | rd | 1100111 | JALR |
| imm[12\|10:5] | rs2 | rs1 | 000 | imm[4:1\|11] | 1100011 | BEQ |
| imm[12\|10:5] | rs2 | rs1 | 001 | imm[4:1\|11] | 1100011 | BNE |
| imm[12\|10:5] | rs2 | rs1 | 100 | imm[4:1\|11] | 1100011 | BLT |
| imm[12\|10:5] | rs2 | rs1 | 101 | imm[4:1\|11] | 1100011 | BGE |
| imm[12\|10:5] | rs2 | rs1 | 110 | imm[4:1\|11] | 1100011 | BLTU |
| imm[12\|10:5] | rs2 | rs1 | 111 | imm[4:1\|11] | 1100011 | BGEU |
| imm[11:0] | | rs1 | 000 | rd | 0000011 | LB |
| imm[11:0] | | rs1 | 001 | rd | 0000011 | LH |
| imm[11:0] | | rs1 | 010 | rd | 0000011 | LW |
| imm[11:0] | | rs1 | 100 | rd | 0000011 | LBU |
| imm[11:0] | | rs1 | 101 | rd | 0000011 | LHU |
| imm[11:5] | rs2 | rs1 | 000 | imm[4:0] | 0100011 | SB |
| imm[11:5] | rs2 | rs1 | 001 | imm[4:0] | 0100011 | SH |
| imm[11:5] | rs2 | rs1 | 010 | imm[4:0] | 0100011 | SW |
| imm[11:0] | | rs1 | 000 | rd | 0010011 | ADDI |
| imm[11:0] | | rs1 | 010 | rd | 0010011 | SLTI |
| imm[11:0] | | rs1 | 011 | rd | 0010011 | SLTIU |
| imm[11:0] | | rs1 | 100 | rd | 0010011 | XORI |
| imm[11:0] | | rs1 | 110 | rd | 0010011 | ORI |
| imm[11:0] | | rs1 | 111 | rd | 0010011 | ANDI |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0000000 | shamt | | rs1 | 001 | rd | 0010011 | SLLI |
| 0000000 | shamt | | rs1 | 101 | rd | 0010011 | SRLI |
| 0100000 | shamt | | rs1 | 101 | rd | 0010011 | SRAI |
| 0000000 | rs2 | | rs1 | 000 | rd | 0110011 | ADD |
| 0100000 | rs2 | | rs1 | 000 | rd | 0110011 | SUB |
| 0000000 | rs2 | | rs1 | 001 | rd | 0110011 | SLL |
| 0000000 | rs2 | | rs1 | 010 | rd | 0110011 | SLT |
| 0000000 | rs2 | | rs1 | 011 | rd | 0110011 | SLTU |
| 0000000 | rs2 | | rs1 | 100 | rd | 0110011 | XOR |
| 0000000 | rs2 | | rs1 | 101 | rd | 0110011 | SRL |
| 0100000 | rs2 | | rs1 | 101 | rd | 0110011 | SRA |
| 0000000 | rs2 | | rs1 | 110 | rd | 0110011 | OR |
| 0000000 | rs2 | | rs1 | 111 | rd | 0110011 | AND |
| 0000 | pred | succ | 00000 | 000 | 00000 | 0001111 | FENCE |
| 0000 | 0000 | 0000 | 00000 | 001 | 00000 | 0001111 | FENCE.I |
| 000000000000 | | | 00000 | 000 | 00000 | 1110011 | ECALL |
| 000000000001 | | | 00000 | 000 | 00000 | 1110011 | EBREAK |
| csr | | | rs1 | 001 | rd | 1110011 | CSRRW |
| csr | | | rs1 | 010 | rd | 1110011 | CSRRS |
| csr | | | rs1 | 011 | rd | 1110011 | CSRRC |
| csr | | | zimm | 101 | rd | 1110011 | CSRRWI |
| csr | | | zimm | 110 | rd | 1110011 | CSRRSI |
| csr | | | zimm | 111 | rd | 1110011 | CSRRCI |

Not in CS61C